Lab 6: Decision Tree

CS 412

This lab can be conducted individually or in groups.

In this lab, you will learn how to build a decision tree, and to visually represent decision making.

Deadline: **23:59, April 7**.

Please refer to `Lab_Guideline.pdf` in the same Google Drive folder as this Jupyter notebook; the guidelines there apply to all the labs.

```
# Let's first import some modules for this experiment
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

np.random.seed(1)
```

Decision trees are among the most powerful Machine Learning tools available today and are used in a wide variety of real-world applications. They are intuitive and easy to interpret. The final decision tree can explain exactly why a specific prediction was made, making it very attractive for operational use. In this section, we will explore how to implement a decision tree from scratch.

The dataset we will use for this problem is the `Banknote` dataset. The task is to predict whether a banknote is authentic given a number of measures taken from a photograph. It is a binary classification problem. The dataset is stored in `data_banknote_authentication.csv` file, which contains 1372 samples(rows), 5 features (column 1-5) for each sample, and the last column (column 6) is the corresponding label.

1. Implementing decision tree from scratch

(88 points)

In this lab, we only consider decision trees represented as a *binary* tree, i.e., each node can have

1. no child, i.e., being a terminal node,
2. two children that are both terminal nodes;
3. one child being a terminal node and the other being an internal node (i.e., has its own children);
4. two children which are both internal nodes.

In general, it is possible to allow three or more children, but for simplicity, this lab only considers two children.

A node represents a single input feature (attribute) and a split value on that feature, assuming the feature is numeric. The leaf nodes (we will call them as terminal nodes) contain an output class (0 or 1) which is used to make a prediction. Once created, a tree can be navigated for a new example by following the branching criteria in each internal node, until reaching a final prediction.

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is recursive binary splitting, where all the features and different split thresholds are tried using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. **All input features and all possible split thresholds are evaluated, and the combination of (feature, threshold) that minimizes the cost is chosen.** Then the dataset is divided into two subsets, and the same selection procedure is used in each of the two subsets, hence called **greedy**.

Note we say "threshold" because all the features in the `banknote` dataset are numeric (i.e., continuously valued). When a feature is discrete, splitting can be done based on dividing all possible values into two categories, e.g., {apple, orange} for the left, and {banana, kiwi, peach} for the right.

In this lab, the **Gini cost function** is used which indicates how *pure* a node is. A node's purity refers to the diversity of labels among all the training examples belonging to the node. Splitting continues until the number of

training examples in a node falls below a threshold or **a maximum tree depth** is reached.

1.1 Gini index (**14 points**)

This lab will use the Gini index as the cost function to evaluate splits. Every step of node construction requires splitting a datalist (a subset of training examples), which is based on one input feature and one value for that feature as the splitting threshold. It can be used to divide training examples into two groups of examples. In particular, examples whose feature value is less than the threshold form the left group, and the rest examples form the right group.

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split. A perfect separation results in a Gini score of 0, where each group contains only one class of examples. In contrast, the worst case is when each group contains 50/50% of both classes, leading to a Gini score of 0.5 (for a 2 class problem). Assume we have m groups of data after splitting ($m = 2$ in this lab). The Gini index for group j ($j = 1$ or 2) can be expressed as follows:

$$g_j = 1 - \sum_{i=1}^{n} P_{ij}^2$$

where P_{ij} is the probability of a sample being classified to class i. Specifically, it can be computed by counting:

$$P_{ij} = \frac{\#\text{ examples of class i in group j}}{\#\text{ examples in group j}}$$

The final Gini score can then be computed by weighted sum over all groups' Gini indices.

$$G = \sum_{j=1}^{m} w_j g_j, \quad \text{where} \quad w_j = \frac{\#\text{ examples in group j}}{\#\text{ examples in the datalist}}.$$

To better demonstrate the formula, let's go through an example step by step. Assume we have split the data into 2 groups:

Group 1 contains **3** samples: **2** positive and **1** negative. Group 2 contains **4** samples: **2** positive and **2** negative. Then we can compute the Gini index

for each group:

$$g_1 = 1 - \left[(\frac{1}{3})^2 + (\frac{2}{3})^2\right] = \frac{4}{9} g_2 = 1 - \left[(\frac{1}{2})^2 + (\frac{1}{2})^2\right] = \frac{1}{2}$$

The final Gini score can be computed by :

$$G = \frac{3}{7} \times g_1 + \frac{4}{7} \times g_2 = \frac{10}{21}$$

In the following code block, impelment a function `gini_score` to compute the Gini score of two given groups.

```
def gini_score(groups, classes):
  '''
  Inputs:
  groups: 2 lists of examples. Each example is a list, where th
  classes: a list of different class labels (it's simply [0.0,
  Outputs:
  gini: gini score, a real number
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  group_one, group_two = groups
  group_1_size = len(group_one)
  group_2_size = len(group_two)

  count_group1 = dict()
  count_group2 = dict()

  for example in group_one:
    class_of_example = example[-1]
    if class_of_example in count_group1:
      count_group1[class_of_example] += 1
    else:
      count_group1[class_of_example] = 1

  for example in group_two:
    class_of_example = example[-1]
    if class_of_example in count_group2:
      count_group2[class_of_example] += 1
    else:
      count_group2[class_of_example] = 1

  p1 = 0
```

```
  for i in count_group1:
    p1 += ((count_group1[i] / group_1_size) ** 2)
  g1 = 1 - p1

  p2 = 0
  for i in count_group2:
    p2 += ((count_group2[i] / group_2_size) ** 2)
  g2 = 1 - p2


  w1 = group_1_size / (group_1_size + group_2_size)
  w2 = group_2_size / (group_1_size + group_2_size)

  gini = (w1 * g1) + (w2 * g2)

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  return gini




'''
unit test:
group1 = [[4.8, 3.1, 1],
[5.4, 3.4, 1],
[7.0, 3.2, 0],
[6.4, 3.2, 0]]
group2 = [[6.0, 3.0, 1],
[5.0, 3.4, 1],
[5.2, 3.5, 0]]
classes = [0, 1]
result = gini_score((group1, group2), classes)
print(result)
'''

'''
should print: 0.47619047619047616
'''

group1 = [[4.8, 3.1, 1],
[5.4, 3.4, 1],
[7.0, 3.2, 0],
[6.4, 3.2, 0]]
group2 = [[6.0, 3.0, 1],
[5.0, 3.4, 1],
```

```
[5.2, 3.5, 0]]
classes = [0, 1]
result = gini_score((group1, group2), classes)
print(result)
```

```
0.47619047619047616
```

1.2 Create split (**6 points**)

Splitting a dataset means dividing a dataset into two lists of examples given a feature and a splitting threshold for that feature. Once we have the two groups, we can then use the above Gini score function to evaluate the cost of the split. Splitting a dataset involves iterating over all examples, checking if its feature value is below or above the split threshold, and assigning the example to the left or right group respectively.

In the following code block, implement a function `create_split`, which splits the given data list into two groups (left and right) for a given feature index and split threshold. **Each `group` is nothing but a list of examples**.

```
def create_split(index, threshold, datalist):
  '''
  Inputs:
  index: The index of the feature used to split data. It starts
  threshold: The threshold for the given feature based on which
        If an example's feature value is < threshold, then it g
        Otherwise (>= threshold), it goes to the right group.
  datalist: A list of samples.
  Outputs:
  left: List of samples
  right: List of samples
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  left = list()
  right = list()

  for data in datalist:
    if data[index] < threshold:
      left.append(data)
    else:
      right.append(data)
```

```
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return left, right

'''
unit test:
index = 1
threshold = 3.4
datalist = [[4.8, 3.1, 1.6, 1],
[5.4, 3.4, 1.5, 1],
[7.0, 3.2, 4.7, 0],
[6.4, 3.6, 2.7, 0]]
result = create_split(index, threshold, datalist)
print(result)
'''

'''
should print: ([[4.8, 3.1, 1.6, 1], [7.0, 3.2, 4.7, 0]], [[5.4,
'''

index = 1
threshold = 3.4
datalist = [[4.8, 3.1, 1.6, 1],
[5.4, 3.4, 1.5, 1],
[7.0, 3.2, 4.7, 0],
[6.4, 3.6, 2.7, 0]]
result = create_split(index, threshold, datalist)
print(result)
```

```
([[4.8, 3.1, 1.6, 1], [7.0, 3.2, 4.7, 0]], [[5.4, 3.4, 1.5,
1], [6.4, 3.6, 2.7, 0]])
```

1.3 Find the best split (12 points)

With the above `gini_score` and `create_split` functions, we now have everything needed to evaluate the splits.

Given a list of data, we must check **every feature and every possible value of the feature in the datalist** as a candidate split, evaluate the cost of the split, and find the best possible split. Here we use the Gini index as the cost, and a lower value is better. Once the best split is found, we can use it as a node in our decision tree.

As an important note on the terminology, both `internal node` and

`terminal node` are collectively referred to as `node`, and are hence both subject to the maximum depth constraint. Both of them are different from `group`, which is just a list of examples.

A terminal node will be directly represented by a class value (e.g., 0 or 1 as a floating point, or depending on how the dataset represents its labels).

An internal/non-terminal node is represented by a dictionary of five fields:

1. `index`: the index of the feature selected to split the node into two groups;
2. `value`: the threshold of the feature by which the node is split;
3. `groups`: the result of `create_split`, which encodes the left and right groups. By running "left_g, right_g = node['groups']", one can retrieve the two groups. Each group of data is its own small data list of just those examples assigned to the left or right group by the splitting process.
4. `left`: a node that represents the left child. It can be either a terminal node or an internal node.
5. `right`: analogous to `left`.

In the following code block, implement a function `get_best_split` to find the best split for the given data list. Return a dictionary (i.e., an internal node) whose `index`, `value`, and `groups` are populated, i.e., storing the index of the chosen feature, the chosen splitting threshold, and the resulting two groups. Leave `left` and `right` unspecified.

```
def get_best_split(datalist):
  '''
  Inputs:
  datalist: A list of samples. Each sample is a list, the last
  Outputs:
  node: A dictionary contains 3 key value pairs, such as: node
  Pseudo-code:
  for index in range(#feature): # index is the feature index
    for example in datalist:
      use create_split with (index, example[index]) to divide d
      compute the Gini index for this division
  construct a node with the (index, example[index], groups) tha
```

```
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  number_of_features = len(datalist[0]) - 1

  unique_classes = set()

  for example in datalist:
    unique_classes.add(example[-1])

  min_gini = float('inf')

  node = dict()

  for index in range(number_of_features):
    for example in datalist:
      groups = create_split(index, example[index], datalist)
      new_gini = gini_score(groups, unique_classes)
      if new_gini < min_gini:
          min_gini = new_gini
          node['index'] = index
          node['value'] = example[index]
          node['groups'] = groups


  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  return node


'''
unit test:
datalist = [[4.8, 3.1, 1.6, 0.3, 1],
[5.4, 3.4, 1.5, 1.4, 1],
[7.0, 3.2, 4.7, 1.4, 0],
[6.4, 3.2, 2.7, 1.5, 0]]
result = get_best_split(datalist)
print(result)
'''

'''
should print:
{'index': 0,
 'value': 6.4,
 'groups': ([[4.8, 3.1, 1.6, 0.3, 1], [5.4, 3.4, 1.5, 1.4, 1]],
'''
```

```
datalist = [[4.8, 3.1, 1.6, 0.3, 1],
[5.4, 3.4, 1.5, 1.4, 1],
[7.0, 3.2, 4.7, 1.4, 0],
[6.4, 3.2, 2.7, 1.5, 0]]
result = get_best_split(datalist)
print(result)
```

```
{'index': 0, 'value': 6.4, 'groups': ([[4.8, 3.1, 1.6, 0.3,
1], [5.4, 3.4, 1.5, 1.4, 1]], [[7.0, 3.2, 4.7, 1.4, 0], [6.4,
3.2, 2.7, 1.5, 0]])}
```

1.4 Build a tree （44 points)

The construction of a decision tree consists of three major components:

1. Recursive splitting.
2. Termination condition of recursion: when to stop splitting
3. Building a tree.

Step 1: termination function (8 points)

For simplicity, we will start with the termination condition, and then move on to the recursion. Two hyperparamters are important here:

- **Maximum Tree Depth**. This is the maximum number of predecessors (parent, grandparent, etc) that each node (terminal or internal/non-terminal) can have. Once the maximum depth of the tree is met, we must stop splitting a node. Deeper trees are more complex and are more likely to overfit the training data. Note we call the root node to have depth 1 (not 0), its child nodes with depth 2, and so on.
- **Minimum Node Size**. This is the minimum number of training examples that a node can contain. If a node is split into two groups and one of them falls below this minimum size, then we should stop splitting that group further, i.e., we should make that group a terminal node. Nodes that account for too few training examples are expected to be too specific and are likely to overfit the training data.

These two parameters will be specified by the user as input arguments of our tree building procedure. There is one more situation. It is possible that

a split is chosen where all examples belong to one group, while the other group is empty. In this case, we will be unable to continue the splitting and should stop. Overall, there are 3 different **stopping conditions** which are clearly illustrated by the following pseudo-code (see the detailed explanations below).

```
'''
left_g, right_g = N['groups']

if either left_g or right_g is empty:
    Set both N['left'] and N['right'] to a terminal node encodi
    return

# check for max depth
if depth of N >= max_depth - 1:   # use >= instead of == in cas
    N['left'] = a terminal node encoding the most common label
    N['right'] = a terminal node encoding the most common label
    return

# process left child
if the number of examples in left_g <= min_size:
    N['left'] = a terminal node encoding the most common label
else:
    N['left'] = get_best_split(left_g)
    build a tree on N['left'] (use recursion)

# process right child similarly
'''
```

```
"\nleft_g, right_g = N['groups']\n \nif either left_g or
right_g is empty:\n    Set both N['left'] and N['right'] to a
terminal node encoding the most common label of the examples
in N\n    return\n \n# check for max depth\nif depth of N >=
max_depth - 1:   # use >= instead of == in case max_depth =
1\n    N['left'] = a terminal node encoding the most common
label in left_g\n    N['right'] = a terminal node encoding the
most common label in right_g\n    return\n \n# process left
child\nif the number of examples in left_g <= min_size:\n
N['left'] = a terminal node encoding the most common label of
in left_g\nelse:\n    N['left'] = get_best_split(left_g)\n
build a tree on N['left'] (use recursion)\n \n# process right
child similarly\n"
```

Here N is the current node which is represented by a dictionary specified in Section 2.1.3. The best split has already been found for N using `get_best_split`, and the fields `index`, `value`, and `groups` have already been computed for N (but not yet for `left` and `right`). Note that upon the completion of running the pseudo-code, the fields `left` and `right` of N will have been populated, which correspond to the left and right child nodes, respectively.

Note that we check "if either left_g or right_g is empty" before checking the max depth, because if either the left or the right group is empty, then N will become a terminal node returning its majority class, hence must satisfy the depth check. However, since we are already "inside" of N, we cannot change it into a terminal node per se; turning N into a terminal node can only be accomplished by N's own parent node (see the recursion in Step 2 below). As a result, we introduced a trick of adding a left child and a right child for N, both being a terminal node returning the majority class of N. Strictly speaking, it is possible that the depth of N is already `max_depth`, which makes the left and right child nodes exceeding `max_depth`. However, since this is just a workaround to represent that N is a terminal node, it does not make any difference in learning and prediction.

In the following code block, implement a function `to_terminal` that returns the most common class value in a group. This will be used to make predictions.

```
def to_terminal(group):
  '''
  Input:
    group: A list of examples. Each example is a list, whose la
  Output:
    label: the label indicating the most common class value in
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  if not isinstance(group, list) or not group:
    return -1

  label_count = dict()
  for example in group:
    class_label = example[-1]
```

```
    if example[-1] in label_count:
      label_count[class_label] += 1
    else:
      label_count[class_label] = 1

  label = max(label_count, key = lambda k: label_count[k])

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  return label

'''
unit test:
group = [[4.8, 3.1, 1],
[5.4, 3.4, 1],
[7.0, 3.2, 0]]
result = to_terminal(group)
print(result)
'''

'''
should print: 1
'''

group = [[4.8, 3.1, 1],
[5.4, 3.4, 1],
[7.0, 3.2, 0]]
result = to_terminal(group)
print(result)
```

1

Step 2: Recursive splitting (30 points)

Building a decision tree involves calling the above functions `get_best_split` and `to_terminal` over and over again on the groups created for each node. Once a node is constructed, we can construct its child nodes recursively on each group of data from the split by calling the same node construction function again.

In the following code block, implement a function `recursive_split` to conduct this recursive procedure. It takes a `node` as an argument as well as the maximum depth, minimum size in a node and the depth of `node`. Before invoking `recursive_split`, three key-value pairs `index`, `value`,

and `groups` have already been computed for `node` using `get_best_split`, i.e., the best split has already been found and stored.

In fact, all you need to do is to implement the pseudo-code in "Step 1: termination function". Here is the step-by-step explanation:

1. The two groups of data carried by the given `node` are extracted as left and right data lists for use. Then delete the `groups` field in `node` to save space because it will not be needed any more. This deletion is not reflected in the pseudo-code, and you need to implement it.

2. Next, we check if either the left group (`left_g`) or the right group (`right_g`) is empty. If so, create a terminal node for both left and right groups by applying `to_terminal` on the non-empty group. Yes, even for the empty group, we also adopt the result of `to_terminal` on the (other) non-empty group. Think why.

3. We then check the depth. If `max_depth` is reached by the child node (i.e., the depth of `node` itself has reached `max_depth` -1), then create terminal nodes for both the left and the right groups using `to_terminal`.

4. Then we process the left child. If the left group's size is below `min_size`, then create a terminal node for it using `to_terminal`. Otherwise, create a split for the left group by `get_best_split`, and assign the resulting node to the `left` field of `node`. Then pass `node[left]` to the `recursive_split` function with 1 + the depth of `node`.

5. The right child is then processed in the same manner.

```
# Create child splits for a node or make terminal
def recursive_split(node, max_depth, min_size, depth):
  '''
  Inputs:
  node:  A dictionary contains 3 key value pairs, node =
         {'index': integer, 'value': float, 'groups': a tuple c
  max_depth: maximum depth of the tree, an integer
  min_size: minimum size of a group, an integer
  depth: tree depth for current node
  Output:
  no need to output anything, the input node should carry its c
```

```
    '''
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    left, right = node['groups']

    del node['groups']

    if not left or not right:
      if not left:
        node['left'] = to_terminal(right)
        node['right'] = to_terminal(right)
        return
      node['left'] = to_terminal(left)
      node['right'] = to_terminal(left)
      return

    if depth >= (max_depth - 1):
      node['left'] = to_terminal(left)
      node['right'] = to_terminal(right)
      return

    if len(left) <= min_size:
      node['left'] = to_terminal(left)
    else:
      node['left'] = get_best_split(left)
      recursive_split(node['left'], max_depth, min_size, depth +

    if len(right) <= min_size:
      node['right'] = to_terminal(right)
    else:
      node['right'] = get_best_split(right)
      recursive_split(node['right'], max_depth, min_size, depth +


    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


'''
should print:
{'index': 2,
 'value': 2.6,
 'left': {'index': 0,
          'value': 4.8,
          'left': 0,
          'right': {'index': 0,
```

```
                    'value': 4.8,
                    'left': 1,
                    'right': 1}},
  'right': {'index': 1,
            'value': 3.2,
            'left': {'index': 0,
                     'value': 4.9,
                     'left': 0,
                     'right': 0},
            'right': {'index': 0,
                      'value': 5.7,
                      'left': 1,
                      'right': 0}}}
'''


node = {'index': 2, 'value': 2.6,
        'groups': ([[4.8, 3.4, 1.9, 0.2, 1],
                    [6.0, 3.0, 1.6, 1.2, 1],
                    [5.2, 3.5, 1.5, 0.6, 1],
                    [4.8, 3.1, 1.6, 0.3, 1],
                    [5.4, 3.4, 1.5, 1.4, 1],
                    [4.3, 3.5, 1.6, 0.6, 0]],
                   [[5.0, 3.4, 4.6, 1.9, 1],
                    [5.2, 3.4, 3.4, 1.5, 1],
                    [8.7, 3.2, 5.6, 0.2, 1],
                    [7.0, 3.2, 4.7, 1.4, 0],
                    [6.4, 3.2, 2.7, 1.5, 0],
                    [4.9, 3.1, 4.9, 1.5, 0],
                    [4.5, 2.3, 4.0, 0.3, 0],
                    [6.5, 2.8, 2.6, 1.5, 0],
                    [5.7, 3.8, 4.5, 1.3, 0],
                    [4.9, 2.4, 3.3, 1.0, 0]])}
max_depth = 4
min_size = 3
depth = 1
recursive_split(node, max_depth, min_size, depth)
print(node)
```

```
{'index': 2, 'value': 2.6, 'left': {'index': 0, 'value': 4.8,
'left': 0, 'right': {'index': 0, 'value': 4.8, 'left': 1,
'right': 1}}, 'right': {'index': 1, 'value': 3.2, 'left':
{'index': 0, 'value': 4.9, 'left': 0, 'right': 0}, 'right':
{'index': 0, 'value': 5.7, 'left': 1, 'right': 0}}}
```

Step 3: Build a tree (6 points)

We can now put all of the pieces together. In the following code block, implement a function `build_tree` to build a decision tree by the given training set, maximum depth, minimum size. We first create the root node by calling `get_best_split`. Then call `recursive_split` to build out the tree with the current depth set to 1. Upon completion of `recursive_split`, the root node should carry the whole tree.

```
def build_tree(train, max_depth, min_size):
  '''
  Inputs:
    - train: Training set, a list of examples. Each example is
    - max_depth: maximum depth of the tree, an integer (root ha
    - min_size: minimum size of a group, an integer
  Output:
    - root: The root node, a recursive dictionary that should c
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  root = get_best_split(train)
  recursive_split(root, max_depth, min_size, 1)

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  return root

'''
unit test:
train = [[4.8, 3.4, 1.9, 0.2, 1],
[6.0, 3.0, 1.6, 1.2, 1],
[5.0, 3.4, 4.6, 1.9, 1],
[5.2, 3.5, 1.5, 0.6, 1],
[5.2, 3.4, 3.4, 1.5, 1],
[8.7, 3.2, 5.6, 0.2, 1],
[4.8, 3.1, 1.6, 0.3, 1],
[5.4, 3.4, 1.5, 1.4, 1],
[7.0, 3.2, 4.7, 1.4, 0],
[6.4, 3.2, 2.7, 1.5, 0],
[4.9, 3.1, 4.9, 1.5, 0],
[4.5, 2.3, 4.0, 0.3, 0],
[6.5, 2.8, 2.6, 1.5, 0],
[5.7, 3.8, 4.5, 1.3, 0],
[4.3, 3.5, 1.6, 0.6, 0],
[4.9, 2.4, 3.3, 1.0, 0]]
```

```
max_depth = 4
min_size = 3
root = build_tree(train, max_depth, min_size)
print(root)
'''

'''
should print:
{'index': 2,
 'value': 2.6,
 'left': {'index': 0,
          'value': 4.8,
          'left': 0,
          'right': {'index': 0,
                    'value': 4.8,
                    'left': 1,
                    'right': 1}},
 'right': {'index': 1,
           'value': 3.2,
           'left': {'index': 0,
                    'value': 4.9,
                    'left': 0,
                    'right': 0},
           'right': {'index': 0,
                     'value': 5.7,
                     'left': 1,
                     'right': 0}}}
'''


train = [[4.8, 3.4, 1.9, 0.2, 1],
[6.0, 3.0, 1.6, 1.2, 1],
[5.0, 3.4, 4.6, 1.9, 1],
[5.2, 3.5, 1.5, 0.6, 1],
[5.2, 3.4, 3.4, 1.5, 1],
[8.7, 3.2, 5.6, 0.2, 1],
[4.8, 3.1, 1.6, 0.3, 1],
[5.4, 3.4, 1.5, 1.4, 1],
[7.0, 3.2, 4.7, 1.4, 0],
[6.4, 3.2, 2.7, 1.5, 0],
[4.9, 3.1, 4.9, 1.5, 0],
[4.5, 2.3, 4.0, 0.3, 0],
[6.5, 2.8, 2.6, 1.5, 0],
[5.7, 3.8, 4.5, 1.3, 0],
[4.3, 3.5, 1.6, 0.6, 0],
```

```
[4.9, 2.4, 3.3, 1.0, 0]]
max_depth = 4
min_size = 3
root = build_tree(train, max_depth, min_size)
print(root)
```

```
{'index': 2, 'value': 2.6, 'left': {'index': 0, 'value': 4.8,
'left': 0, 'right': {'index': 0, 'value': 4.8, 'left': 1,
'right': 1}}, 'right': {'index': 1, 'value': 3.2, 'left':
{'index': 0, 'value': 4.9, 'left': 0, 'right': 0}, 'right':
{'index': 0, 'value': 5.7, 'left': 1, 'right': 0}}}
```

1.5 Prediction (12 points)

To make predictions with a decision tree, we need to navigate the tree for each test example.

In the following code block, implement a function `predict` to predict the label for all test examples. For each test example, the navigation of the tree can be implemented by using a while loop, or as a recursive function where the same prediction routine is called recursively based on the `index` and `value` fields of the internal nodes, until a terminal node is reached.

In the function, we need to check if a child node is a terminal value (class label to be returned as the prediction), or an internal node that is represented as a dictionary. You may find the function isinstance useful.

```
# Make a prediction with a decision tree
def predict(root, sample):
  '''
  Inputs:
  root: the root node of the tree. a recursive dictionary that
  sample: a list
  Outputs:
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  while isinstance(root, dict):
    if sample[root['index']] < root['value']:
      root = root['left']
    else:
      root = root['right']
```

```
    return root

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

'''
unit test:
root = {'index': 2, 'value': 2.6, 'left': {'index': 0, 'value':
sample1 = [5.4, 3.4, 1.5, 1.4]
sample2 = [4.3, 3.5, 1.6, 0.6]
print(predict(root, sample1))
print(predict(root, sample2))
'''
'''
should print:
1
0
'''

root = {'index': 2, 'value': 2.6, 'left': {'index': 0, 'value':
sample1 = [5.4, 3.4, 1.5, 1.4]
sample2 = [4.3, 3.5, 1.6, 0.6]
print(predict(root, sample1))
print(predict(root, sample2))
```

```
1
0
```

2. Training and testing (12 points)

Now we are ready to apply this algorithm to the `Banknote` dataset. In the following code block, evaluate the decision tree model as follow:

- Load the dataset to a list of examples (already done)
- Since the loaded data features are of string type, covert all features to float type
- Split the dataset into a training set and a test set. Use the first 1000 samples for training, and the rest for testing.
- Build a tree by providing the traning set, maximum depth and minimum size.
- Make prediction for all test examples using the built tree.

- Print the accuracy and f1 score for the test set.

```
# load and prepare data
import pandas as pd
import urllib.request
import shutil
from csv import reader
from random import seed

url = 'https://www.cs.uic.edu/~zhangx/teaching/data_banknote_au
file_name = 'data_banknote_authentication.csv'
with urllib.request.urlopen(url) as response, open(file_name, '
    shutil.copyfileobj(response, out_file)

file = open(file_name, "rt")
lines = reader(file)

df = pd.read_csv(file_name,
                 sep='\t',
                 header=None)
df.head()
```

	0
0	3.6216,8.6661,-2.8073,-0.44699,0
1	4.5459,8.1674,-2.4586,-1.4621,0
2	3.866,-2.6383,1.9242,0.10645,0
3	3.4566,9.5228,-4.0112,-3.5944,0
4	0.32924,-4.4552,4.5718,-0.9888,0

```
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
seed(1)

dataset = list(lines)
max_depth = 6
min_size = 10
num_train = 1000

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
for example in dataset:
  for index in range(len(example)):
    example[index] = float(example[index])

training_data = dataset[:num_train]
test_data = dataset[num_train:]

root = build_tree(training_data, max_depth, min_size)

test = list()
for example in test_data:
  test.append(example[:len(example) - 1])

pred = list()
for example in test:
  pred.append(predict(root, example))

actual = list()
for example in test_data:
  actual.append(example[-1])

acc = accuracy_score(actual, pred)
f1 = f1_score(actual, pred)

print("Accuracy:\t", acc)
print("f1_score:\t", f1)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
Accuracy:    0.9543010752688172
f1_score:    0.9766162310866575
```

Submission Instruction

You're almost done! Take the following steps to finally submit your work.

1. After executing all commands and completing this notebook, save your `Lab_6.ipynb` as a PDF file, named as `X_Y_UIN.pdf`, where `X` is your first name, `Y` is your last name, and `UIN` is your UIN. Make sure that your PDF file includes all parts of your solution, including the plots.

- Print out all unit test case results before printing the notebook into a PDF.
- If you use Colab, open this notebook in Chrome. Then File -> Print -> set Destination to "Save as PDF". If the web page freezes when printing, close Chrome and reopen the page. If Chrome doesn't work, try Firefox.
- If you are working on your own computer, we recommend using the browser (not jupyter) for saving the PDF. For Chrome on a Mac, this is under *File->Print...->Open PDF in Preview*. When the PDF opens in Preview, you can use *Save...* to save it.
- Sometimes, a figure that appears near the end of a page can get cut. In this case, try to add some new lines in the preceding code block so that the figure is pushed to the beginning of the next page. Or insert some text blocks.

2. Upload `X_Y_UIN.pdf` to Gradescope under `Lab_6_Written`.

3. A template of `Lab_6.py` has been provided. For all functions in `Lab_6.py`, copy the corresponding code snippets you have written into it, excluding the plot code. **Do NOT** copy any code of plotting figures and do not import **matplotlib**. This is because the auto-grader cannot work with plotting. **Do NOT** change the function names.

4. Zip `Lab_6.py` and `Lab_6.ipynb` (**2 files**) into a zip file named `X_Y_UIN.zip`. Suppose the two files are in the folder `Lab_6`. Then zip up the **two files inside the `Lab_6` folder**. **Do NOT zip up the folder `Lab_6`** because the auto-grader cannot search inside a folder. Submit this zip file to Gradescope under `Lab_6_Code`.

5. The autograder on Gradscope will be open all the time. We designed some simple test cases to help you check wehther your functions are executable. You will see the results of running autograder once you submit your code. Please follow the error messages to debug. Since those simple test cases are designed for debugging, it does not guaranttee your solution will work well on the real dataset. It is your responsibility to make your code logically correct. Since all functions are tested in batch, the autograder might take a few minutes to run after submission.

If you *only* try to get real-time feedback from auto-grader, it will be fine to

just upload `Lab_6.py` to `Lab_6_Code`. However, the final submission for grading should still follow the above point 4.

You can submit to Gradescope as often as you like. We will only consider your last submission before the deadline.